Engineering a Media Search Engine

By Brian Cullinan

12/16/2010

## Introduction

Currently, the search engine Google does not offer a search specifically for music. Our project aims to alleviate this issue by scanning popular RSS feeds from websites such as Reddit.com. These results are then scanned for audio and video files and embedded objects that play relevant music. In addition to compiling many music results from multiple feeds, the results are ranked based on popularity contributed by multiple metrics.

The initial design used a single server to scan web pages and pull out interesting information such as page titles, mime-types, links, and media content. A starting point is fed in to the scanner; this starting point is closely related to a “hub” of information for a particular genre of music. Other interesting information includes tags associated with each media link, and also the community generated popularity. Once enough training data is retrieved statistical information for the most popular media, tags, and relational information can be generated to provide accurate search results.

In the second iteration of implementing the project, the goal changed from displaying the results most closely related to the term entered in the search (most relevant tags), to displaying the most popular media and collecting as much media as possible from many different feeds. To accomplish this, the system was redesigned to support multiple inputs from multiple “scanning” servers to a single data-warehouse. The data-warehouse is then used to display the relevant and most popular search results.

The front-end design has remained the same; a simple, easy to use Google style search box is presented. Terms related to the artist, genre, or any other tag can be entered to display the relevant results, automatically sorted by popularity.



Related Work and Background

There are not many fully featured implantations of this type of search engine. Google does not yet have a search engine specific for music, although since most of the results are accompanied by videos, Google’s video search is the closest thing to providing media content related to genre specific search terms. However, this project takes advantage of community contributed ratings as well as popularity determined using HITS.

### Popularity

There is a lot of work done in the field of data-mining. In this specific case, we have used multiple different metrics to determine popularity. One particular metric is how many pages link to each media record. This metric is normalized and used in conjunction with other metrics; this will be discussed in more detail later. This link counting is closely related to the HITS algorithm, which is also similar to the way Google ranks its results. The HITS algorithm was developed by Kleinberg.

Another important ranking mechanism used is “Wisdom of the Crowds”. Using existing popularity ratings contributed by many different sources is a key component of the ranking system. Our project uses different metrics including the rating system provided by YouTube results to train the system and provide more accurate ranking in our search. For example, if many people who are interested in a particular genre rank a song very highly, is it statistically shown that subsequent users interested in that genre will find that song entertaining.

### Technologies Used

A common way to mine out interesting information from a web page is to use Regular Expressions. These are used frequently in this project to pull out links. The most important usage of regular expressions is for pulling out links to every page and scanning them for more and more links recursively. Regular expressions can also be used to match attributes sent in the return header when categorizing each link.

Reading RSS feeds is also a key part of this project. The “scanner” is responsible for inputting all the links from RSS feeds, and these feeds each have their own “update time”, that is used for telling scanners like mine when to check back to retrieve new items in the feed.

Aspell is a popular open source library for spell checking. This library is used in conjunction with PHP code to provide spell checking just like Google. Google has done a lot of research on what makes a good user interface, and this design is mimicked in our project.

jQuery is a key component of the search result updating process that gives the site a “live” feel. When new search results are contributed to the data warehouse, they are automatically stacked on to the top of the page. These results are not necessarily the most popular, but they give the user some insight into trends in the particular genre they have searched for, giving them a broader set of results and possibly even providing the user with new music that they haven’t heard before.

Related tags are provided on the left hand side. This uses the TF-IDF algorithm to provide a list of tags that are usefully related to the input query. The section on text mining was used for reference while implementing this feature.



Methodology and Design

The first task of this project was to collect a lot of training data to use for future filtering of links and skipping sections of websites that were likely to not contain any music related content. This process must be repeated due to time and memory constraints, and it must also be updated due to the constantly changing internet trends. This task is accomplished by saving the recorded links in a database and using a “cron” job running at a relatively small interval to update the set of links. Each page scanned is scanned recursively, all the links found on one page are then also scanned for links, and those links are scanned, and so on. This is done 3 levels deep from the starting point, which surprisingly can return hundreds and thousands of results, since each level is increased by a factor of n where n is the number of links found on the previous level.

The next task is to go through every link that is not a link to a text page and “classify” the media as audio/video/embedded content as well as the “tags” that are closely related to that media. This task takes quite a bit of time, and therefore running multiple servers and a using a data-warehouse is very useful. This task is based on the time each link is scanned and as it is processed the time for the record is updated. This task also runs as a cron job, so it will be repeated at a frequent interval on the web server.

The ware-housing script is run least often, but still often enough on multiple servers that the user can interact with freshly scanned search results. The ware-house script goes through every record, pulling out the most popular items and their corresponding tags, and does a lot of counting on the database. It checks the total number of documents that each tag occurs in, in order to compute the TF-IDF values for the tags. It also discards all page articles from the ware-house as they cannot be played by the user and should not be used in the search results. This process maximizes the usefulness of the scanned data and search results. This feature was added on too late in the project and therefore these filtered results won’t be used in generating the statistical information in the next section.

The last task was to create a way for general users to interact with the collected data. The first approach did a simply find and match on the database and all the columns. Results were initially organized in order of most “relevant”, which was done by calculating the number of occurrences of a particular tag or phrase within a record and also adding the distances from the entered phrase to the length of each column in the record. For example, a record with a title “dubstep dance” would rank higher in the results than “a dubstep example”, because the distance between the first item and the search term “dubstep” is much shorter than the search term and the second example.

In future revisions the search became more and more dependent on other metrics as they were added such as the number of relevant tags and the TF-IDF of those tags. The data-warehouse was also added after the initial functionality of having cron jobs to update the database and the front end.

The front end user interface provides a search box for users to enter a genre or any tag that can be found in the database. As new results are scanned, AJAX and jQuery is used to update the page, showing new results related to their search terms. When an item is selected for playing it “sticks” to the top of the page and new results appear underneath the currently playing item. This is useful from a user interface perspective because you don’t want your currently playing item to constantly be moving down the page while you are trying to watch it. When a new item is selected for play the currently playing item is “replaced” back in to the page from where it came from. The embedded objects are loaded from Youtube or Soundcloud on the fly, instead of trying to load every object with the page and hiding them, they are loaded when an item is clicked.

Spell check was added in the most recent version of the project. Spelling is done by comparing each word in the search terms against a dictionary, then using the Aspell suggested words and regular expressions to match misspelled words and make them bold and italicized.

Implementation:

Here is some sample code of PHP creating the tags IDF values and adding them to the data-warehouse.

$tags[] = $tag;

$tag\_results = db\_assoc('SELECT count(\*) FROM media WHERE LEFT(Tags, ?)=? OR LOCATE(?, Tags) > 0', array(

strlen($tag),

$tag,

$tag,

));

$idf = log((1 + count($result)) / $tag\_results[0]['count(\*)']);

As you can imagine, this will take a long time to generate when every tag must be read from every entry in the database and then it is checked if it is already added to the data-warehouse. On a database with only a few hundred thousand records, the time it takes to generate all the IDF values is about 10 minutes.

Every query to the database is “parameterized” which means the values to insert in to the query are kept separate from the actual query string, and the database library cleans all of the input completely preventing SQL injection. This is very useful for security reasons.

This is an example of a fetch call, which performs a download of a remote page for processing by the link reader:

$page = fetch($url, array(), array('header\_callback' => 'handle\_default\_read'));

This type of function call is a key component of the API. This robust fetch() function is what allows the system to process remote pages and pull out all links and media content-type information. It basically allows the script to act like a web crawler scanning for links recursively. When “header\_callback” is used, it can check the header content-type and if it is for example type “audio/mp3” it can classify the content and then cancel the request before it tries to download the entire file.

Here is the full code for the web crawler functionality:

function scan\_url($url, $depth = 3, $\_current\_depth = 0)

{

$page = fetch($url, array(), array('header\_callback' => 'handle\_default\_read'));

if($page['status'] == 200)

{

// verify the page is valid

if(substr($page['headers']['content-type'], 0, 5) == 'text/')

{

// update page title

if(preg\_match('/<title[^>]\*?>(.\*?)<\/title>/i', $page['content'], $matches) > 0)

$result = db\_query('UPDATE media SET Name=?, MIME=? WHERE URL=?', array($matches[1], $page['headers']['content-type'], $url));

// get all the hyperlinks in a page

$links = get\_links($page['content'], $url);

foreach($links as $i => $link)

{

// remove the fragment from the link

if(strpos($link, '#') !== false)

$link = substr($link, 0, strpos($link, '#'));

// add to pages to scan database

$pages = db\_assoc('SELECT \* FROM media WHERE URL=? AND Source=?', array($link, $url));

if(count($pages) > 0)

continue;

$config = array(

'URL' => $link,

'Source' => $url,

);

$id = db\_insert('INSERT INTO media ' . sql\_insert($config), array\_values($config));

// scan url recursively

if($\_current\_depth < $depth)

scan\_url($link, $depth, $\_current\_depth + 1);

}

}

elseif(robots\_allowed($url))

{

handle\_url($url);

}

}

// update url with time in pages

db\_query('UPDATE media SET Added=? WHERE URL=?', array(date('Y-m-d h:i:s'), $url));

}

This function is recursive, and scans every page 3 levels deep by default from the starting “hub”. This may be too high and could probably be reduced to a depth of 2 since most of the links on the “hub” lead directly to a Youtube or Soundcloud page.

This function is used mainly by the crawler to pull out all the links off of the page. The links are then “cleaned” by removing the fragment of the link, so if a page links to some other page with a “bookmark” the entire page will be scanned and the pages link will be recorded without the bookmark, this prevents multiple links with different bookmarks to the same page from being recorded.

// read all the links off a page

function get\_links($content, $url)

{

$link\_count = preg\_match\_all('/<a[^>]\*?href=([\'"])([^\1]\*?)\1/i', $content, $matches);

$links = array();

if($link\_count > 0)

{

// sort out only valid links

foreach($matches[2] as $i => $link)

{

$validated\_link = generic\_validate\_url(array('link' => $link), 'link');

if($validated\_link)

$links[] = trim(htmlspecialchars\_decode(get\_full\_url($url, $validated\_link)));

}

}

return array\_unique($links);

}

These are the most critical components of the system, but there are many more not listed here. A brief description of the key files are listed here:

Scanner.php – this is the link crawler, it is given a starting page and craws websites recursively for hyperlinks. This is run about every hour to update the links to pages with media.

Cron.php – this is also run by cron, but much more frequently, about every 10 minutes. This goes through every link in the database checking the content type and pulling out embedded objects.

Index.php – This is the main interface to the search engine, this is the only page presented to the user.

Database.inc – This is the main database handler, this contains all the database related functionality including connecting to multiple databases and provides some utility functions for properly formatting your database queries to be compatible with the ADOdb library which does the parameterized queries.

Seach.php – This is the main searching functionality. This file contains all the utility functions for creating a database search query. This also checks the spelling and returns any corrections that were made.

Handlers.php – This is probably the most important file for actually loading the information in to the database. This extendable design dynamically calls handle\_() functions based on the URL that is loaded from the page, for example, a function signature could be handle\_www\_youtube\_com and all URLs will be passed to that function for processing the Title of the Youtube video, whereas another link may be passed to the function handle\_soundcloud\_com for processing. This can be extended by adding new functions for different sites, this is useful for controlling how each different embedded player is actually played in the list using Javascript, or pulling out more accurate ranking information from specific sites. This process of dynamically calling a function is very similar to Java’s introspection capabilities.

Results

A good metric to use for determining what music is popular would be the top ten lists for large record labels of the specific genre. Beatport.com provides a top 10 lists and API for getting track and rating information. This list could be used to determing if the search results are ranking items with good popularity values. Initial comparisons show that there are very few tracks in common with the search engine and the top 10 lists. However, by looking at the artists present, the search engine does manage to return some artists in common with the top 10 lists. About 5 percent of the results artists appear on the top 10 lists from Beatport. Doing a Google search and checking for the number of results for a particular song or artist could also be used to determine what music is most popular. Since the popularity is used from Reddit (the starting hub), and the rating system is used from Youtube and Soundcloud, these sites need to be compared against other metrics, however they are still a valid indication if the scanner is interpreting the popularity properly. Reddit offers a “top” page which could be used to determine the MOST popular music and how the system ranks the relevant results.



Possible Extensions and Future Work

A lot more work could be done to improve the “popularity” of search results. This may be accomplished by improving the classifier, and figuring out what people are listening to using other metrics. The project currently uses HITS and community contributed ranking and rating systems, but a system could be added to tie together artists and albums from other sites like Discogs and Musicbrainz, or just using the popularity of an artist to rank their other tracks and related tracks.

The system does not currently start playing each individual player automatically; it requires user intervention for each song to be played. This could be improved greatly by adding a really smart Javascript to automatically call the players API play() functions in order.

The spell check needs to be improved to not correct the names of actual genres. Since new genres are being invented all the time, it should leave very popular tags “uncorrected”. However, this may just be a problem with the dictionary.

Gathering more statistical information about the ranking system and determining if it is working properly is an essential step in any future work done on this project.

Conclusions

In conclusion, I would say the project was a partial success. The interface turned out extremely well, and mimics the simple interface of Google real-time search. The scanning capabilities are robust and can capture tons of information; however the scanners are somewhat slow trying to look up links. Since a lot of checks are just checking to see if a link has already been scanned a much better hashing function could be used that just returns true or false if a link exists in the hash table, such as Google SparseHash. This algorithm can store hundreds of thousands of entries in a very small amount of space such as 10 MB of memory. Additionally, it performs hash lookups in constant O(1) time. This would be a perfect solution for improving the speed of the simple checks for duplicate link entries.

References

1. <https://code.google.com/p/google-sparsehash/>
2. <https://secure.wikimedia.org/wikipedia/en/wiki/RSS>
3. <https://secure.wikimedia.org/wikipedia/en/wiki/GNU_Aspell>
4. <https://secure.wikimedia.org/wikipedia/en/wiki/JQuery>
5. <https://secure.wikimedia.org/wikipedia/en/wiki/Ajax_%28programming%29>
6. <https://secure.wikimedia.org/wikipedia/en/wiki/HITS_algorithm>
7. <http://www.amazon.com/Data-Mining-Concepts-Techniques-Management/dp/1558609016>
8. <https://secure.wikimedia.org/wikipedia/en/wiki/The_Wisdom_of_Crowds>
9. <https://secure.wikimedia.org/wikipedia/en/wiki/CURL>
10. <https://secure.wikimedia.org/wikipedia/en/wiki/Cron>

Appendix

Instructions for executing the program:

1. Install a PHP server.
2. Download PHP ADOdb and put in your includes directory
3. Configure the database connecting in the settings.php file
4. Set up cron to execute the scanner.php and cron.php at selected intervals, as well as ware-house.php
5. Connect to <http://yourserver.com/index.php>

Here is a brief explanation of many of the references to Wikipedia pages above:

SparseHash – A hash table algorithm invented by Google that takes up very little space per entry and is searchable in O(1) constant time.

RSS – Really simple syndicate, a popular form of XML for updating readers on new media related content.

Aspell – An open source dictionary library

jQuery – A Javascript technology that uses CSS style “selectors”

HITS - Hyperlink-Induced Topic Search

cURL – Connect URL, a library that acts very similar to wget to retrieving pages on the internet and analyzing headers and page content

cron – A linux task scheduling service